探索的仕様記述のための履歴ツールの提案と実装

小田 朋宏
株式会社 SRA
tomohiro@sra.co.jp

張 漢明
南山大学
chang@nanzan-u.ac.jp

山本 恭裕
公立はこだて未来大学
yxy@acm.org

中小路 久美代
公立はこだて未来大学
kumiyo@acm.org

荒木 啓二郎
熊本高等専門学校
araki@kyudai.jp

要旨

ソフトウェア開発の過程を困難なものとしている一つの要因として，複雑なものを対象にした試行錯誤が挙げられる．ソフトウェアシステムを作成する過程において，複雑な対象ドメインやソフトウェアシステム自身への理解を探索的に深め，得られた知見を整理し，ドキュメントとして記録することが求められる．ソフトウェア開発における履歴の管理と整理は，ソフトウェア工学の古くからの課題であり，多くのツールが実現されて広く利用されている．本論文では，ソフトウェア開発における試行錯誤の中でも，形式仕様記述における個人作業の中での探索的な作業に注目して，仕様記述の編集や表現式の評価実行のような細粒度の履歴の整理のための手法を提案する．そして，実行可能な仕様記述言語 VDM-SL での仕様記述における探索的試行を記録し，後に仕様記述者自身がその過程を整理し，記録するためのツールを提案し，その実装を紹介する．

1. はじめに

ソフトウェア開発を困難なものにしている要因として，複雑さがある．ソフトウェアシステム自体の複雑さに加え，ソフトウェアシステムの対象ドメインの複雑さや，ソフトウェアシステムを利用する人の振る舞いに関連した複雑さから，開発の開始時点では開発に必要な知識を前もって獲得することは困難である．ソフトウェアシステムを作りながら，対象ドメインに存在する概念や，ユーザーの振る舞いから利用シナリオを抽出し，かつ，それら対象ドメインや利用シナリオに対してどのようなソフトウェアシステムが適合するか理解を深めていく必要があることから，多くのソフトウェア開発では探索的な試行によって理解を深めていくことが求められる．

機能仕様は開発対象の機能や特性を記述したものであり，他の工程の多くで参照される．機能仕様の策定にも探索的な試行が行われるが，試行には部分的な失敗を含むことが多いことから，他の工程に波及するとコストがかかる．形式仕様はソフトウェアシステムを実装する以前にその特性を分析することを可能にすることから，仕様記述段階での探索的試行に適している．著者らは形式仕様工程のうち探索的試行が多くおこなわれる初期段階を探索的仕様記述として，それを支援するための開発環境 ViennaTalk [1] を開発してきた．ViennaTalk は仕様記述言語として VDM-SL [2] を採用し，その実行可能なサブセットによる仕様アニメーションを中心にして，VDM-SL 仕様記述を実行可能なプロトタイプとして様々な利用シナリオへの適合性を評価し，探索的な試行を支援するための機能を提供している．

探索の結果として得られる知見は，最終的に記述された VDM-SL 仕様だけではない．探索の過程で試した多くの暫定的な仕様記述や，その上で検討した利用シナリオや，その利用シナリオでのシステムの動作を記録することで，その仕様記述の機能項目がなぜそのように定義されているかを説明するドキュメンテーションとしての役割を果たすことができる．

本論文では，探索的な試行の履歴を整理するためのモデルと，その実装例として VDM-SL 用のウェブ IDE で

ある VDMPad [1] に探索的試行の履歴を整理する機能を追加した VDMPad-EpiLog を紹介し，その利用例を示す．第 2 節で探索的試行の履歴を整理するための構造とそれに対する操作を提示し，その実装として VDMPad-EpiLog の機能とその利用例を第 3 節に示す．第 4 節では関連する既存のツールとして github および git を中心に比較検討する．第 5 節に議論とまとめを示す．

2. VDM-SL 仕様記述の履歴モデル

本節では，VDM-SL 仕様を記述する個人作業における探索的試行の履歴を整理するための構造を提案する．本研究では開発における探索的試行について，対象に関する部分的な知識に基づいて暫定的仮説的なモデルの記述を作成し分析を行うことを 1 単位の試行とした上で，1 つの試行を通して深められた理解に基づいて次の施行を扱う，サイクルプロセスと捉える．仕様記述をおこなう中で，探索的に繰り返された試行を抽出することで，履歴を構造化し整理する．

VDM-SL は，実行可能なサブセットを持つ，モデル規範型の形式仕様記述言語である．仕様記述者は対象を内部状態および内部状態に対する操作としてモデル化し，データ型，定数，関数，状態空間，操作の 5 種類の定義によって記述する [2]．データ型を `types` 部で宣言し，定数および関数をそれぞれ `values` 部および `functions` 部で定義することによって，そのシステムが扱う概念とその関係を記述する．VDM-SL では定数および関数は参照透明である．`state` 部では，システムが持つ状態空間を定義するために，静的に型付けられた変数のリスト，状態に対する不変条件，および初期状態を宣言する．システムが提供する操作を `operations` 部で定義する．操作は状態に対する参照や変更を行うことができる．

形式仕様記述言語としての VDM-SL の大きな特徴として，そのサブセットをインタプリタで実行可能であることが挙げられる．VDMTools や Overture Tool [3] を始めとする VDM-SL 開発環境の多くが，REPL (Read-Eval-Print Loop) インタプリタを提供し，仕様記述者は編集中の仕様記述に対してインタプリタを起動して，表現式を入力して評価実行することができる．仕様記述の探索的試行において，インタプリタ実行は編集中の仕様記述の特性を把握する上で，重要な役割を果たしている．

VDM-SL によるモデル化の前段階として，システムの機能仕様に対する原始要求として，システムの利用シー

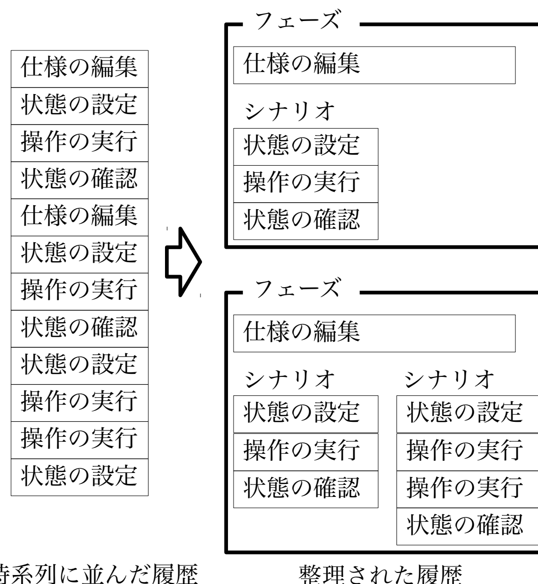

図 1. 履歴整理の例

ンをいくつか想定し，それぞれの利用シーンをユースケースとして，どのような操作をどのような順番でどのような判断を伴いながら実行するかを想定することが多い．システムの動作を表現した記述として，ユニットテスト [4] がある．

VDM-SL 仕様に対するユニットテストフレームワークは vdmUnit や viennaUnit [5] などが既に開発され利用されている．

ユニットテストでは，システムの構成部品に対して想定される利用シナリオ毎に，テストケースを定義する．各テストケースは，シナリオの前提となるシステム状態の設定，システムに対する一連の操作の実行，操作結果の確認を，テスト用の操作として記述したものである．VDM-SL 仕様のユニットテストは，記述対象に対する利用シナリオへの適合性の検査を自動化したものと見なすことができる．

本研究では，ユニットテストを参考にして，探索的試行のモデルを定義した．探索的試行の中では，仕様記述の編集，状態の編集，操作の評価実行が行われる．仕様記述者は，それぞれの時点での暫定的な仕様記述に対して，利用シナリオをいくつか想定して，利用シナリオを状態編集と操作列として模擬的に実行し，実行結果が利用シナリオが想定した通りかを確認する．履歴を，仕様記述のバージョンに対応する**フェーズ**，利用シナリオに

対応する**シナリオ**，個々の状態編集や操作実行に対応する**ラン**の 3 つの単位にまとめることとした．

フェーズは，記述対象に関する理解の変化を捉えることを主眼にした単位である．仕様記述は，仕様記述者が対象をモデル化したものであり，仕様記述者による対象に対する理解を表現している．本節冒頭で述べた通り，本研究では仕様記述における探索を，仕様記述者による対象への理解を表現した暫定的仮説的な仕様記述とその記述に対する一連の検証を 1 つの試行とする．フェーズは 1 つの仕様記述とそれに対する検証をおこなうための一連の評価実行をまとめたものであり，1 単位の試行と見なすことができる．

シナリオは，仮説検証における検証の単位である．仕様記述で定義されたシステムの動作が対象ドメインや原始要求に対して妥当であるかどうかを確認するために，システムの特定の動作を切り出したものをシナリオとする．1 つのフェーズの中で，複数のシナリオによる妥当性の確認が行われる．シナリオには，原始要求に含まれる利用シナリオや対象ドメインからくるものもあれば，仕様記述に定義された機能単位からくるものもある．

ランは，シナリオの妥当性を確認するために行う，仕様記述者による操作の単位である．VDM-SL 仕様はシステムを状態空間と操作群により定義することから，VDM-SL 仕様記述での探索では，状態の初期化や，操作の評価実行を試行の基本単位として扱うことができる．具体的にどのような種類のランがあるかは，開発環境のユーザインタラクションの設計による．REPL (Read-Eval-Print Loop) インタプリタの場合，インタプリタが提供するコマンドのうち，アニメーション実行に影響を与えるものをランとして扱うことができる．Overture Tool には，仕様記述が定義する操作をボタンにした簡易 GUI によるアニメーション機能があるが，この場合は各ボタンの押下をランとして扱うことができる．

図 1 に，仕様の修正をともなう探索的試行の履歴について，時系列からフェーズ，シナリオ，ランの 3 つの粒度の構造に整理した例を示す．左側には，探索の中で行われた仕様の編集，状態の設定，操作の実行，状態の確認が上から下の順に時系列で列挙されている．仕様記述における探索的試行では，暫定的な仕様記述を作成し，その仕様記述が利用シナリオにどのように適合するか，システムがどのように動作するかを確認する．いくつかの利用シナリオが試され，システムの動作が仕様記述者の意図と異なる場合には仕様記述を編集する．1 つの暫定的な仕様記述に対する一連の履歴項目を，1 つのフェーズとする．フェーズは，履歴の時系列から，仕様記述の編集から，次の仕様記述の編集までの区間を切り出すことで，容易に抽出することができる．

フェーズの中では，いくつかの利用シナリオに関する適合性の確認が行われる．ここでの利用シナリオは，原始要求の中で明示的に示されたものもあるが，仕様記述者によるドメインへの理解から仮説的に想定しているものもある．したがって，シナリオの同定は必ずしも機械的にできるとは限らない．探索の履歴を整理する上で，シナリオは原始要求や仕様記述者の理解を反映する重要な単位であることから，探索の履歴を整理するためのツールを設計する上で重要なポイントになる．

ランはシナリオを構成する基本要素であり，シナリオの内容はランの時系列として表現される．ランは多くの場合，開発環境が収集した時系列の履歴の各項目から抽出することができる．

3. VDMPad-EpiLog

本節では，探索的試行の履歴を整理するツールの実装例として VDMPad-EpiLog を説明する．VDMPad-EpiLog は， 探索的な仕様記述の特に初期段階の試行を想定して設計されたウェブ IDE である VDMPad [1] に対する拡張として実装された．

図 2 に VDMPad の画面例を示す．VDMPad は仕様アニメーションを中心に設計されている．仕様アニメーションとは，仕様をインタプリタ等で評価実行することで，与えられた仕様を実装したシステムがどのように動作するかをシミュレートする技術である．VDMPad は，VDM-SL で記述された仕様のソースと，仕様が定義する状態空間におけるシステムの現在の状態を保持し，それをプロトタイプとしてウェブページ上に表示する．仕様記述者はプロトタイプに対して，仕様記述，状態，評価式を編集して与え，評価実行ボタンを押すことで評価結果を得るとともに，プロトタイプの状態を更新する．VDMPad は，柔軟な探索のために，仕様 / 状態 / 評価式のいずれも常に編集可能であり，それらはいずれもプロトタイプへの修正操作として扱われる．また，プロトタイプの仕様を編集しても，プロトタイプを再起動することなく，継続してプロトタイプに対する操作を行うことができる．

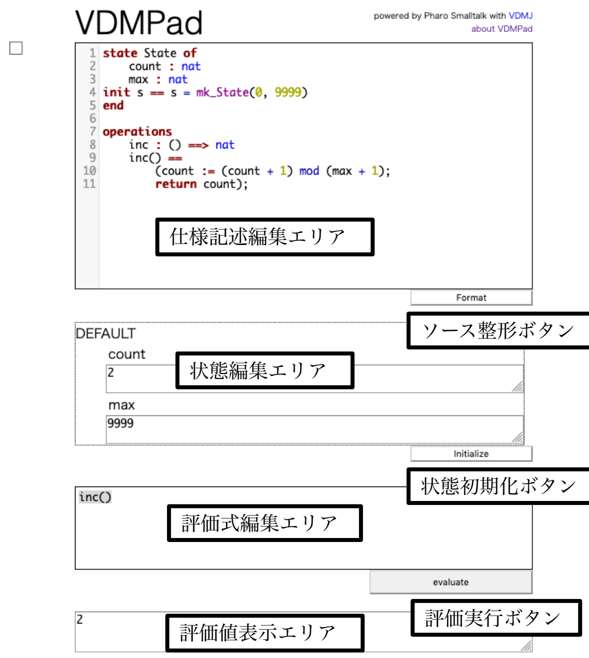

図 2. VDMPad の画面例

3.1. VDMPad-EpiLog の概要

VDMPad-EpiLog は，VDMPad を拡張して，履歴を整理する機能を追加したものである．VDMPad では，プロトタイプは受動的であり，仕様記述者が評価実行ボタンおよび状態初期化ボタンを押すことで，プロトタイプを駆動する．評価実行ボタンが押されると，VDMPad は仕様記述のソース，現在の状態，評価式をサーバに送信し，評価結果の値，評価後の状態，エラーの場合にはエラーメッセージを返す．VDMPad-EpiLog はそれを捉えて，エラーでない場合のみ，仕様記述のソース，評価前の状態，評価式，評価結果の値，評価後の状態を保存し，以下の手順で自動的にフェーズ，シナリオ，ランとして整理する．

- 仕様の断絶：仕様記述が直前のものと異なる場合には，新しいフェーズを生成する．
- 状態の断絶：上の条件に当てはまらず，かつ，評価前の状態が直前の履歴での評価後の状態と異なる場合には，新しいシナリオを生成し，最新フェーズのシナリオリストに追加する．

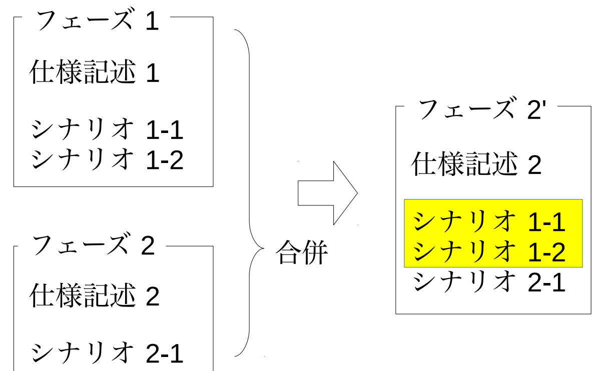

図 3. フェーズの合併

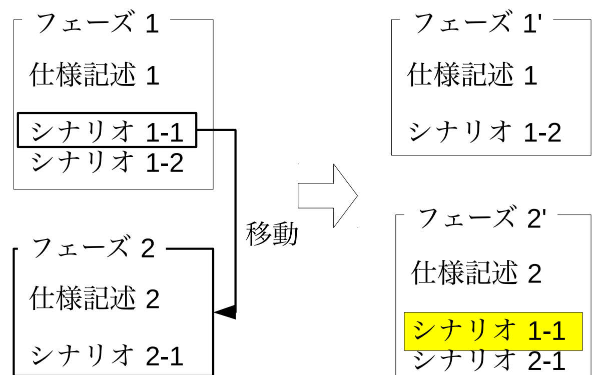

図 4. シナリオの移動

- 連続した実行：いずれの条件にも当てはまらない場合には，最新シナリオの末尾にランとして追加する．

こうして自動的に整理された履歴は，ユーザによる探索的試行を実行の時系列に忠実に反映した履歴である．しかし探索の結果得られた知見としてまとめる上では，いくつかの問題点が考えられる．

仕様記述を編集するごとにそれに対応するフェーズが生成されるが，そうしたフェーズにはあまり重要ではないものも多い．仕様記述の修正のうち，知見として残す必要のない軽微な修正は，重要なフェーズと合併する，あるいは削除することで，重要なフェーズのみを残すことができる．図 3 にフェーズの合併を示す．フェーズ 1 の仕様に軽微な問題があり，フェーズ 2 の仕様でそれが解決された場合，フェーズ 1 をフェーズ 2 に合併することで両方のシナリオを修正後であるフェーズ 2 の仕様

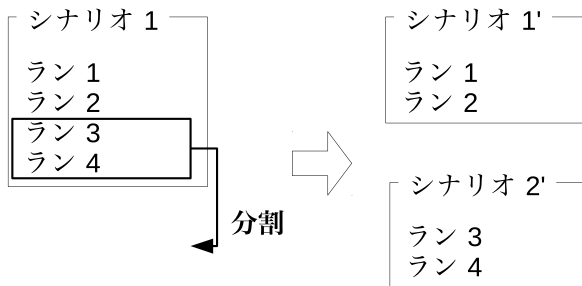

図 5. シナリオの分割

の上でのシナリオとすることができる．ただし，フェーズ 1 にあった 2 つのシナリオは，試行時にはフェーズ 1 の仕様で実行されたものであり，フェーズ 2 の仕様での実行結果とは異なる可能性がある．したがってフェーズ 1 から移動したシナリオに含まれるランの実行結果は無効マークが付与される．無効とされたランは，移動先のフェーズの仕様で再実行することで，無効マークを取り除くことができる．また，図 4 に示すように，特定のシナリオのみ次のフェーズに移動することもできる．

探索的試行の中で，同一フェーズ内で連続して複数のシナリオを検討する場合，シナリオ開始時に特に状態を変更する必要がない場合がある．状態の変更を伴わずにシナリオでのランを開始すると，VDMPad-EpiLog はシナリオの切れ目を検出できず，同一シナリオの継続として処理する．探索的試行の中で意図したシナリオに分割するために，シナリオの特定のラン以降を別のシナリオとして分割することができる．シナリオの分割操作を図 5 に示す．

フェーズやシナリオ，ランには繰り返し同じものが現れたり，特に重要ではないため知見として残す必要のないものも多い．したがって，不要なフェーズやシナリオやランを削除する操作も提供される．

3.2. VDMPad-EpiLog の UI

本節では VDMPad-EpiLog の UI を説明する．図 6 に VDMPad-EpiLog の画面例を示す．VDMPad のメニュー中の VDMPad-EpiLog ボタンを押すと，従来の VDMPad の右側に構造化された履歴が表示される．ただし，ウェブブラウザの横幅が足りない場合には，従来の VDMPad の下部に表示される．構造化された履歴の表示部分を，本論文では EpiLog 画面と呼ぶことにする．

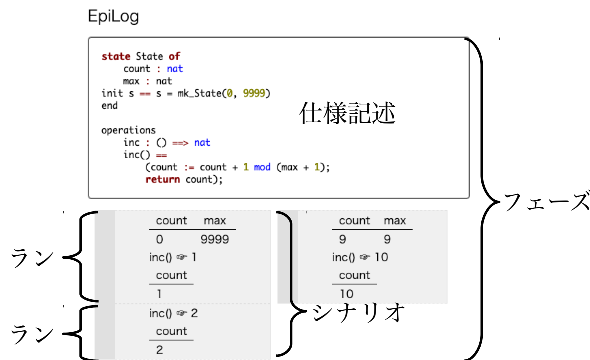

図 6. VDMPad-EpiLog の画面例

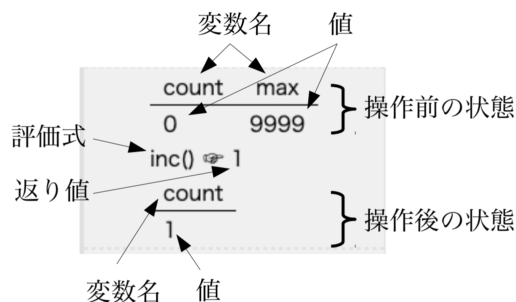

図 7. ランの表示例

EpiLog 画面には，フェーズごとに仕様記述とシナリオが表示される．各フェーズでは上段に黒枠で囲まれた仕様記述の下に，背景色がついたシナリオが列挙される．各シナリオには，左側に太い帯状の濃いグレーが 1 つながりで表示され，その右側にランが上から下に並べて表示される．シナリオ中のランとランの間は，薄い横線で区切られる．

各ランには，上段に実行前の状態，中段には評価実行する表現式とその返り値，下段には実行後の状態が表示される．図 7 にランの表示例を示す．水平に引かれた区切り線の上に変数名，下にその値が表示される．実行前の状態については，当該ランの直前のランの実行後の状態からの差分が表形式で表示される．シナリオ中の多くのランでは，操作前の状態は直前のランの実行後の状態

を引き継ぐため，差分がなく，表示は省略される．ただし，各シナリオの最初のランは，前のランがないため，実行前の状態がそのまま表示される．中段には，右方向を指差したシンボルの左側に表現式，右側にそれを評価した返り値が表示される．下段には，実行前の状態からの実行後の状態の差分が表形式で表示される．

フェーズ，シナリオ，ランに対する修正操作は，操作対象を選択して行う．操作対象を選択すると，その対象に対して実行可能な操作に対応したボタン列が表示されるので，ボタンを押すことで操作を実行することができる．フェーズの選択は，仕様記述の黒枠内をクリックして行う．シナリオの左側の帯をクリックすると，シナリオが選択される．帯の右側のランの上をクリックすると，ランが選択される．

3.3. VDMPad-EpiLog の使用例

本節では，VDMPad-EpiLog の典型的な使用例として，VDMPad 上でのカウンタの仕様記述の探索的試行の例を紹介し，その履歴を整理するステップを示す．記述するカウンタは，自然数のカウント値を 1 ずつ増加させていくものである．ただし，上限値が設定され，上限値を越えると 0 に戻るものとする．カウンタに対する操作として，カウント値を進める inc，カウント値を 0 にリセットする reset，上限値を設定する setMax の 3 つを提供するが，ここでは，まずはカウント値を進める inc を定義するための記述作業を対象とする．他の 2 つの操作 reset および setMax の定義は省略し，VDM-SL 仕様中に記述しない．

図 8 に示した画面では，2 つのフェーズが表示されている．最初のフェーズ (以後，フェーズ 1) では，暫定的な仕様記述を入力し，2 つのシナリオが検討された．最初のシナリオでは，inc を連続して 2 回実行して，カウント値が期待通りに 1 ずつ増加することを確認した．次に，カウント値が上限値を越える場合の動作を確認するために，カウント値 count と上限値 max にそれぞれ 9 を設定し，inc 操作を行った．すると，カウント値が 0 になることが期待されていたところが，10 となった．仕様記述者はこの時点での仕様記述に誤りがあることに気付いた．

仕様中の inc 操作の定義の中で，count + 1 mod (max + 1) とある式は，正しくは (count + 1) mod (max + 1) を意図していた．表現式 count + 1 mod

```
state State of
    count : nat
    max : nat
    init s == s = mk_State(0, 9999)
    end
operations
    inc: () ==> nat
    inc() == (count := count + 1 mod (max + 1); return count);
```

count	max
0	9999

inc() ☞ 1

count
1

inc() ☞ 2

count
2

count	max
9	9

inc() ☞ 10

count
10

```
state State of
    count : nat
    max : nat
    init s == s = mk_State(0, 9999)
    end
operations
    inc: () ==> nat
    inc() == (count := (count + 1) mod (max + 1); return count);
```

count	max
9	9

inc() ☞ 0

count
0

図 8. 整理開始時の画面例

```
state State of
    count : nat
    max : nat
    init s == s = mk_State(0, 9999)
    end
operations
    inc: () ==> nat
    inc() == (count := (count + 1) mod (max + 1); return count);
```

count max
0 9999
inc() ☞ 1
count
1
inc() ☞ 2
count
2

count max
9 9
inc() ☞ 10
count
10

count max
9 9
inc() ☞ 0
count
0

図 9. 軽微な修正のフェーズを合併した画面例

```
state State of
    count : nat
    max : nat
    init s == s = mk_State(0, 9999)
    end
operations
    inc: () ==> nat
    inc() == (count := (count + 1) mod (max + 1); return count);
```

count max
0 9999
inc() ☞ 1
count
1
inc() ☞ 2
count
2

count max
9 9
inc() ☞ 10
count
10

count max
9 9
inc() ☞ 0
count
0

図 10. シナリオを再実行した画面例

(max + 1) は，演算子の結合順位により count + (1 mod (max + 1)) と解釈され，1 mod (max + 1) は常に 1 であることから，inc 操作は上限値に関係なく常にカウント値を 1 ずつ増加させる動作となっていた．仕様記述者はこの誤りに気付き，修正を行い，再びカウント値 count と上限値 max にそれぞれ 9 を設定し，inc 操作を行った．これが 2 つ目のフェーズ (以後，フェーズ 2) であり，実行結果は仕様記述者の意図通り，カウント値が 0 になった．

フェーズ 2 はフェーズ 1 からの軽微な修正であり，ここではフェーズ 2 を中心に探索的試行をまとめることにする．ただし，フェーズ 1 では初期状態から inc 操作を 2 つ行うシナリオを実行しているが，フェーズ 2 では実行していない．本来はフェーズ 2 でも実行すべきであるが，仕様記述者は上限値を超えた時の動作の修正を行っており，他のシナリオは念頭になかった．そこで，フェーズ 1 で検討したシナリオをフェーズ 2 にまとめるため，フェーズ 1 をフェーズ 2 に合併する．

図 9 に合併の結果を示す．合併の結果，フェーズは 1 つのみになった．このフェーズではフェーズ 2 にあった，誤り修正後の仕様記述が使われる．元のフェーズ 1 から合併で移動した 2 つのシナリオに含まれるランは，元の誤りを含む仕様記述で実行されたため，このフェーズの履歴としては正しくない．これら 2 つのシナリオの合わせて 3 つのランは無効であり，無効であることを示すために黄色を背景色にして表示されている．

無効なランは，新しい仕様記述の上で再実行することで，有効にすることができる．図 10 に，1 つ目のシナリオを再実行した結果を示す．初期状態から inc 操作を 2 回実行し，結果は元のフェーズ 1 での実行結果と同一であった．最初の探索的試行の段階では，誤りを含んだ仕様記述での実行を確認しただけであったところが，その履歴をまとめる作業を通して，修正後も同様に意図通りの動作をすることを確認することができた．

次に，カウント値が上限値を超えた場合のシナリオの再実行を行う．カウント値が上限値を超えた場合については，仕様記述を修正する前は意図した通りの動作をしなかった．図 10 では，元の意図した通りの動作ではなかったランが無効化された状態で表示されている．このランを再実行することで，この仕様記述では意図した通りの結果に置き換わることを確認することができる．再実行の結果を図 11 に示す．元のフェーズ 2 で実行されたシナリオと同一の結果であることが確認できる．

カウント値が上限値を超えた場合のシナリオは内容が同一であることから，冗長であり削除することにする．削除することで，一連の探索的試行の整理が完了する．履歴を整理した結果を図 12 に示す．この探索で到達した仕様記述と，その上で検討された 2 つのシナリオが並んで表示されている．1 つ目のシナリオは，初期状態から inc 操作を 2 回連続で行うもので，仕様記述者の意図

```
state State of
    count : nat
    max : nat
    init s == s = mk_State(0, 9999)
    end
operations
    inc: () ==> nat
    inc() == (count := (count + 1) mod (max + 1); return count);
```

```
count  max
0      9999
inc() ☞ 1
count
1
inc() ☞ 2
count
2
```

```
count  max
9      9
inc() ☞ 0
count
0
```

```
count  max
9      9
inc() ☞ 0
count
0
```

図 11. 全シナリオを再実行した画面例

```
state State of
    count : nat
    max : nat
    init s == s = mk_State(0, 9999)
    end
operations
    inc: () ==> nat
    inc() == (count := (count + 1) mod (max + 1); return count);
```

```
count  max
0      9999
inc() ☞ 1
count
1
inc() ☞ 2
count
2
```

```
count  max
9      9
inc() ☞ 0
count
0
```

図 12. 重複シナリオを削除し整理を完了した画面例

通りの動作であることが確認された．2 つ目のシナリオは，カウント値が上限値を越える場合の `inc` 操作を行うもので，これも仕様記述者の意図通りの動作であることが確認された．これら 2 つのシナリオがこの仕様記述で想定された利用シナリオであり，それぞれユニットテストにおけるテストケースとして利用することができる．

4. 関連する既存ツール

ソフトウェア開発を支援するための工学研究では，開発の履歴の保存や集積，追跡性の確立は古くからの課題の 1 つである．現在ソフトウェア開発で多く利用されている統合開発環境では操作履歴に関する機能として undo/redo を提供するものは多いが，履歴に構造を与えて系統的にまとめることは行われていない．大きな粒度での開発の履歴の保存や集積，追跡性の確立は，コードレポジトリおよびそれと統合された開発チームホスティングサービスが提供している．github は現在オープンソースをはじめ広く利用されている開発チームホスティングサービスである．ソースコードレポジトリは分散バージョン管理システム git によりバージョン間の関係を有向グラフとして保存することができる．また，イシュー管理システムおよび修正提案 (pull request) から特定のバージョンへの参照を記録することができることから，あるバージョン差分の意図がどの問題を解決するためのもので，それがどの修正提案として提案され，誰によりソースコードレポジトリに取り入れたかを追跡可能にしている．このことから，github は開発コミュニティにおける長期間での開発の記録を整理し保存し，追跡性を確保した形で集積する機能を持つサービスと言える [6]．ただし，github でのバージョン管理，イシュー管理および修正提案は，開発の中での極めて大きな粒度での試行を捉えるものであり，基本的に個人間の円滑な連携や，新規参入者が文脈を理解しやすくするための仕組みである．本研究が扱うような，個人作業の内部でおこなう 1 つ 1 つの試行的な評価実行を対象にしたものではない．

ローカルな git レポジトリにも，バージョン差分を併合する機能があることから，個人作業での試行の履歴を整理することができる．しかし git レポジトリが扱う対象はソースコードであり，インタプリタ上あるいはコンパイルされた実行ファイルを動作させた履歴は扱わない．ユニットテスティングでのユースケースのバージョン差分を管理することで，各試行で検討したシナリオの変遷

を記録することが可能だが，本研究で扱う極めて短期間での試行とは，扱う試行の粒度が異なると考えられる．

5. 議論とまとめ

探索的な仕様記述で行われる探索的試行の履歴から，探索全体を整理し知見として残すための枠組みとして，フェーズ，シナリオ，ランによる構造化と，構造化された履歴への編集操作を提案した．具体的なツールとして VDMPad-EpiLog を実装し，その利用例を示した．

VDMPad-EpiLog は VDM-SL 仕様を対象として扱うことを前提に設計されている．VDM-SL はモデル規範型の形式仕様記述言語であり，モデルが取りうる状態空間を厳密に規定する．状態空間が明確に規定されていることで，試行における評価実行の文脈を容易に保存することができる．

プログラミング言語でのユニットテスティングでもシナリオを再現するための状態設定を行うが，一般にプログラミング言語では状態空間が明示的でなく，多くの不確定要素を抱える．例えば，ファイル入出力は多くの場合にはバッファリング IO を通して，オペレーティングシステムのシステムコールが提供する共有リソースにアクセスする．他のプロセスとの相互作用も暗黙的に発生する可能性がある．これらはソースコードには明示的に記述されない部分であり，VDMPad-EpiLog が行うような細粒度での再実行を実現するのは容易ではない．シナリオが前提とする状態の再現の容易さと，それがもたらすシナリオに対する編集操作は，モデル規範型の形式仕様記述言語 VDM-SL の利点として挙げることができる．

今後の課題として，より複雑な仕様記述での探索的試行の支援が挙げられる．VDMPad は比較的小規模な仕様記述を想定して設計・実装されているため，複数モジュールに分割された複雑な仕様記述を扱うのに適していない．より複雑化した VDM 仕様を扱うために，VDMPad-EpiLog と同様な機構を ViennaTalk にも実装し，比較評価したい．

関連研究で挙げた科学研究向けノートブック環境への応用も考えられる．状態の再現性の高さ，および，操作実行の結果の検証の容易さは，再現性や検証可能性が要求されるノートブック環境の実現にも有用であることが期待できる．

6. 謝辞

本研究は，JSPS 科研費 (18K18033) および JSPS 科研費 (19K11911) の助成を受けた．有益なご指摘を頂いた査読者の方々に感謝する．

参考文献

[1] 小田朋宏, 荒木啓二郎 “形式仕様工程の初期段階に着目した統合仕様記述環境 ViennaTalk”, コンピュータソフトウェア, 34 巻, 4 号, ppp. 4_129-4_143, 日本ソフトウェア科学会, 2017.

[2] J. Fitzgerald and P. G. Larsen “Modelling Systems – Practical Tools and Techniques in Software Development”, Cambridge University Press, 1998.

[3] P. G. Larsen, N. Battle, M. Ferreira, J. Fitzgerald, K. Lausdahl and M. Verhoef “The Overture Initiative – Integrating Tools for VDM”, *SIGSOFT Softw. Eng. Notes*, Vol. 32, No. 1, pp. 1–6, 2010.

[4] P. Hamill “Unit Test Frameworks”, O ’Reilly, 2004.

[5] 小田朋宏, 荒木啓二郎 “アジリティのある探索的形式仕様記述のためのテストフレームワーク”, ソフトウェアシンポジウム 2018 論文集, 2018.

[6] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb “Social coding in GitHub: transparency and collaboration in an open software repository”, in Proceedings of the ACM 2012 conference on computer supported cooperative work, pp. 1277-1286, 2012.